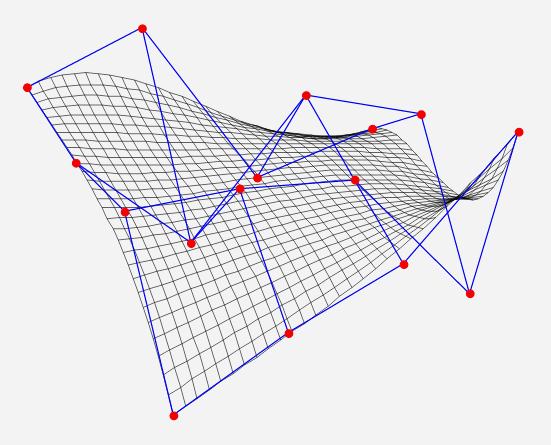
# 计算机图形学

画图应用



学号：0x091395

姓名：uknowho

一、 需求分析：

通过图形学基础实验过程，将功能集中到一个应用程序中，能够分别实现相应图形的绘制。

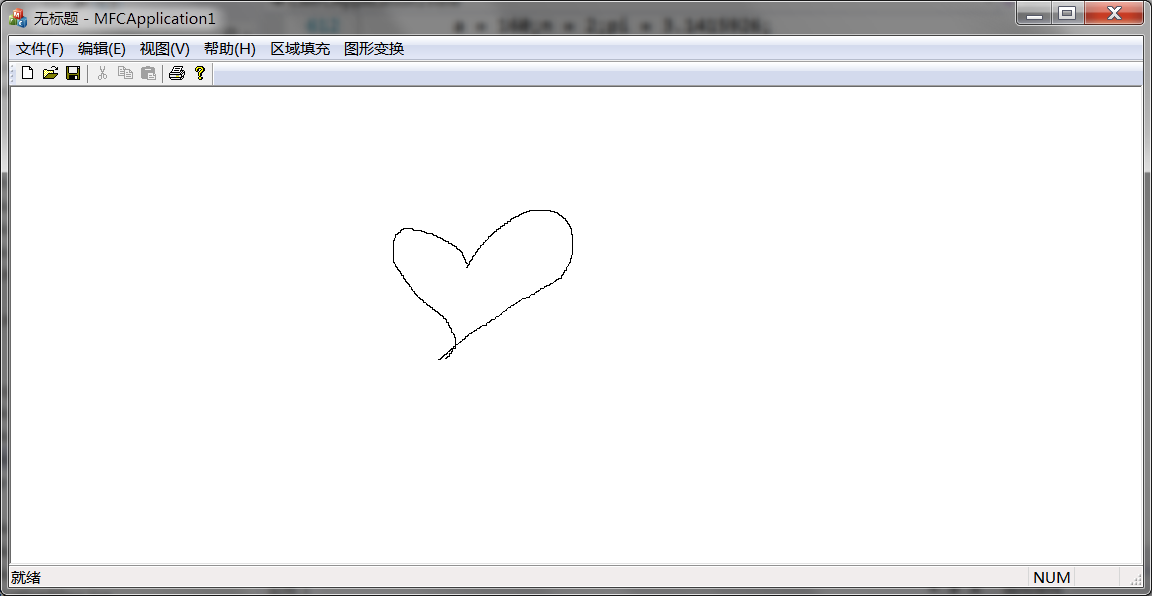
在程序中通过相应算法，实现绘制简单图形，如：直线、圆、椭圆；绘制复杂图形，如：正叶图、蝴蝶结。通过算法实现多边形的裁剪，填充。

通过使用OpenGL绘制多面体，如：正四面体、正八面体、球面、茶壶等。

二、 概要设计：

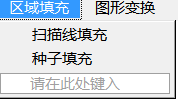
## 1. 界面设计

基于VS2012单文档型为基础。添加相应的选项菜单。



## 2. 主要功能

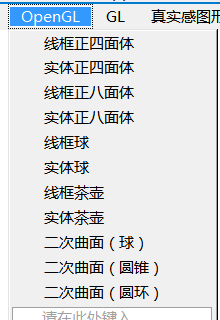
* 画笔
* 填充



* 蝴蝶结
* 正叶线



* 绘制多面体



三、 系统编码：

## 原理及核心代码

1、画点

①原理：通过函数SetPixel将指定坐标处的像素设为指定的颜色，通过此方法实现画点。

②实现：

CCMyDrawDoc\* pDoc = GetDocument(); //获得文档对象指针

pDC->SetPixel(p.x,p.y,p.color);//画点

2、画直线

DDA算法是一个增量算法。增量算法：在一个迭代算法中，每一步的x、y值是用前一步的值加上一个增量来获得。

通过各行各列象素中心构造一组虚拟网格线。按直线从起点到终点的顺序计算直线与各垂直网格线的交点，然后根据误差项的符号确定该列象素中与此交点最近的象素。

圆的Bresenham算法：

以点(0,r)为起点，按顺时针方向生成圆时，相当于在第一象限内，所以 y 是 x 的单调递减函数。y的计算式为：



令d1、d2分别为yi到y、yi-1到y的距离，可知：



令判断式di=d1-d2，并代入d1、d2，则有：



1. 如果di<0，则y=yi，即选择当前像素的正右方作为下一个像素，递推公式为：



（2）如果di≥0，则y=yi-1，即选择当前像素的右下方作为下一个像素，递推公式为：



（3）计算判别式的初值。初始点为（0，R），则：

3、画圆

①原理：本程序使用Bresenham算法画圆。

Bresenham算法：圆的Bresenham算法：

以点(0,r)为起点，按顺时针方向生成圆时，相当于在第一象限内，所以 y 是 x 的单调递减函数。y的计算式为：



令d1、d2分别为yi到y、yi-1到y的距离，可知：



令判断式di=d1-d2，并代入d1、d2，则有：



1. 如果di<0，则y=yi，即选择当前像素的正右方作为下一个像素，递推公式为：



（2）如果di≥0，则y=yi-1，即选择当前像素的右下方作为下一个像素，递推公式为：



②代码实现：

void CMyView::OnBresenhamcircle()

{

CDC \*pDC=GetDC();

int x0=100,y0=100,x,y,r=80,c=0; //黑色圆弧，r半径，x0，y0圆心坐标

float e,d;

e=3-2\*r;x=0;y=r;//设置差值，半径

while(x<=y)

{

if (e<0)

{e=e+4\*x+6;x++;}

else{e=e+4\*(x-y)+10; x++;y--;}

//画点（八个对称点）

pDC->SetPixel(x+x0,y+y0,c);

pDC->SetPixel(-x+x0,y+y0,c);

pDC->SetPixel(-x+x0,-y+y0,c);

pDC->SetPixel(x+x0,-y+y0,c);

pDC->SetPixel(y+x0,x+y0,c);

pDC->SetPixel(-y+x0,x+y0,c);

pDC->SetPixel(-y+x0,-x+y0,c);

pDC->SetPixel(y+x0,-x+y0,c);

}

ReleaseDC(pDC);

}

4、画椭圆

①原理：本程序使用的是椭圆的中点算法。

算法如下：椭圆方程：F(x, y) = b2x2 + a2y2 - a2b2 = 0定义椭圆弧上某点的切线法向量N如下：N（x，y）=2b\*b\*xi+2a\*a\*yj。对方程分别求x偏导和y偏导，最后得到椭圆弧上(x,y)点处的法向量是（2b2x, 2a2y）。椭圆弧法向量的y分量比较大，即：2b2(x + 1) < 2a2(y – 0.5)；椭圆弧法向量的x分量比较大，即：2b2(x + 1) > 2a2(y – 0.5)。y方向每变化1个单位，x方向变化大于一个单位，因此中点算法需要沿着x方向步进画点，x每次增量加1，求y的值。同理，中点算法沿着y方向反向步进，y每次减1，求x的值。假设当前位置是P(xi, yi)，则下一个可能的点就是P点右边的P1(xi+1, yi)点或右下方的P2(xi+1, yi-1)点，取舍的方法取决于判别式di：di = F(xi+1, yi-0.5) = b2(xi+1)2 + a2(yi-0.5)2 – a2b2 若di < 0，表示像素点P1和P2的中点在椭圆内，这时可取P1为下一个像素点。此时xi+1 = xi + 1，yi+1 = yi，

di+1 = di + b2(2xi + 3) 计算出di的增量是b2(2xi + 3)。同理，若di >= 0，表示像素点P1和P2的中点在椭圆外，这时应当取P2为下一个像素点。此时xi+1 = xi + 1，yi+1 = yi - 1，

di+1 = d1 + b2(2xi+3) + a2(-2yi+2) 计算出di的增量是b2(2xi+3)+a2(-2yi+2)。计算di的增量的目的是减少计算量，提高算法效率，每次判断一个点时，不必完整的计算判别式di，只需在上一次计算出的判别式上增加一个增量即可。

②代码实现：

void CCMyDrawView::drawEllispe(CPoint srcPoint,CPoint descPoint, CDC \*pDC)

{

//圆心x坐标，圆心y坐标，长轴长，短轴长

int x0,y0,a,b;

//两种情况

if(descPoint.y>srcPoint.y)

{

x0=descPoint.x-(descPoint.x-srcPoint.x)/2;

y0=descPoint.y-(descPoint.y-srcPoint.y)/2;

a=(descPoint.x-srcPoint.x)/2;

b=(descPoint.y-srcPoint.y)/2;

}

else

{

x0=srcPoint.x-(srcPoint.x-descPoint.x)/2;

y0=srcPoint.y-(srcPoint.y-descPoint.y)/2;

a=(srcPoint.x-descPoint.x)/2;

b=(srcPoint.y-descPoint.y)/2;

}

int color=RGB(0,0,0);

int x,y;

float d1,d2;

x=0;y=b;

d1=b\*b+a\*a\*(-b+0.25);

pDC->SetPixel(x+x0,y+y0,color);

struct mypoint p={x+x0,y+y0,0};

allpoints.push\_back(p);

while (b\*b\*(x+1)<a\*a\*(y-0.5))

{

if (d1<0)

{

d1+=b\*b\*(2\*x+3);

x++;

}

else

{

d1+=(b\*b\*(2\*x+3)+a\*a\*(-2\*y+2));

x++; y--;

}

pDC->SetPixel(x0+x,y0+y,color);

pDC->SetPixel(x0+x,y0-y,color);

pDC->SetPixel(x0-x,y0+y,color);

pDC->SetPixel(x0-x,y0-y,color);

struct mypoint p1={x0+x,y0+y,0};

allpoints.push\_back(p1);

struct mypoint p2={x0+x,y0-y,0};

allpoints.push\_back(p2);

struct mypoint p3={x0-x,y0+y,0};

allpoints.push\_back(p3);

struct mypoint p4={x0-x,y0-y,0};

allpoints.push\_back(p4);

} // 上半部分

d2=(b\*(x+0.5))\*(b\*(x+0.5))+(a\*(y-1))\*(a\*(y-1))-(a\*b)\*(a\*b);

while (y>0)

{

if (d2<0)

{

d2+=b\*b\*(2\*x+2)+a\*a\*(-2\*y+3);

x++; y--;

}

else

{

d2+=a\*a\*(-2\*y+3);

y--;

}

pDC->SetPixel(x0+x,y0+y,color);

pDC->SetPixel(x0+x,y0-y,color);

pDC->SetPixel(x0-x,y0+y,color);

pDC->SetPixel(x0-x,y0-y,color);

struct mypoint p1={x0+x,y0+y,0};

allpoints.push\_back(p1);

struct mypoint p2={x0+x,y0-y,0};

allpoints.push\_back(p2);

struct mypoint p3={x0-x,y0+y,0};

allpoints.push\_back(p3);

struct mypoint p4={x0-x,y0-y,0};

allpoints.push\_back(p4);

} //下半部分

}

5、绘制蝴蝶结

①原理：蝴蝶结顾名思义即类似蝴蝶结的图案。是经过设置二维坐标x，y函数以及利用旋转变换绘制的图案。

②代码实现：

void CMy2DtransView::OnDrawRose()

{CDC \*pDC=GetDC();

//d半径长，x1，y1起点坐标，x2，y2终点坐标

int d,x1,x2,y1,y2;

//圆周率π，

float pi,a,e;

CPen pen;

pen.CreatePen(PS\_SOLID,1,RGB(0xff,0,0));

CPen \*pOldPen = pDC->SelectObject(&pen);

pi = 3.1415926;

d = 80;

for (a = 0; a<=2 \* pi; a+= pi/360)

{

e = d \* (1+0.25\*sin(4\*a));

e = e \* (1 + sin(8\*a));

x1 = int(320+e\*cos(a));

x2 = int(320+e\*cos(a + pi/8));

y1 = int(200+e\*sin(a));

y2 = int(200+e\*sin(a + pi/8));

//画线

pDC->MoveTo(x1,y1);

pDC->LineTo(x2,y2);

}

}

6、绘制正叶图

①原理：通过设置二维坐标x，y函数以及利用旋转变换绘制的图案。

②代码实现：

void CMy2DtransView::OnDrawLeaf()

{

CDC \*pDc=GetDC();

CPen pen;

pen.CreatePen(PS\_SOLID,1,RGB(0,0,0xff));

CPen \*pOldPen = pDc->SelectObject(&pen);

int a,n,cx,cy,gx,gy,flag,k;

double r,p,th,x,y,pi;

a = 160;n = 2;pi = 3.1415926;

cx = 700;

cy = 500;

flag = 0; k = 200;

for (p = 1; p >= 0.2; p-=0.2)

{

for (th = 0; th <= 2\*pi+0.1; th += pi/k)

{

r = fabs(a\*cos(n\*th)\*p);

x = r\*cos(th);

y = r\*sin(th);

gx = int(cx+x);

gy = int(cy+y);

if (flag == 0)

{

pDc->MoveTo(gx,gy);

}

flag = 1;

pDc->LineTo(gx,gy);

}

flag = 0;

}

pDc->SelectObject(pOldPen);

}

7、矩形区域裁剪

①原理：

1、直线段编码裁剪算法（Cohen-Sutherland 算法）

假设窗口是标准矩形，即边与坐标轴平行的矩形，由上（y=wyt）、下（y=wyb）、左（x=wxl）、右（x=wxr）四条边描述。

算法基本思想：

对每条直线段p1(x1,y1)p2(x2,y2)分三种情况处理：

(1) 直线段完全可见，“简取”之。

(2) 直线段完全不可见，“简弃”之。

(3) 直线段既不满足“简取”的条件，又不满足“简弃”的条件，需要对直线段按交点进行分段，分段后重复上述处理。

2、多边形逐边裁剪算法：要保持窗口内多边形的边界部分，而且要将窗框的有关部分按一定次序插入多边形的保留边界之间，从而使剪裁后的多边形的边仍然保持封闭状态。

②代码实现：

int CCMyDrawView::C\_S\_Line(CDC \*pDC, int x1, int y1, int x2, int y2)

{

int code1,code2,code,x,y;

encode(x1,y1,&code1); //(x1,y1)处的编码

encode(x2,y2,&code2); //(x2,y2)处的编码

while (code1!=0||code2!=0) //当 code1 不等于 0 或 code2 不等于 0

{

if ((code1&code2)!=0) return 0; //当 code1 与 code2 不等于 0，在同侧；

code=code1;

if (code1==0) code=code2;

if ((LEFT&code)!=0) //求交点

{

x=WL;

y=y1+(y2-y1)\*(WL-x1)/(x2-x1);

}

else if ((RIGHT&code)!=0)

{

x=WR;

y=y1+(y2-y1)\*(WR-x1)/(x2-x1);

}

else if ((BOTTOM&code)!=0)

{

y=WB;

x=x1+(x2-x1)\*(WB-y1)/(y2-y1);

}

else if ((TOP&code)!=0)

{

y=WT;

x=x1+(x2-x1)\*(WT-y1)/(y2-y1);

}

if (code==code1)

{

x1=x;y1=y;

encode(x,y,&code1);

}

else

{

x2=x;y2=y;

encode(x,y,&code2);

}

}

//end while,表示 code1，code2 都为 0，其中的线段为可视部分

//pDC->MoveTo(x1+350,y1);

//pDC->LineTo(x2+350,y2);

struct mypoint p={x1,y1,0};

allpoints2.push\_back(p);

}

//编码

void CCMyDrawView::encode(int x, int y, int \*code)

{

int c=0;

if (x<WL) c=c|LEFT;

else if (x>WR) c=c|RIGHT;

if (y>WB) c=c|BOTTOM;

else if (y<WT) c=c|TOP;

\*code=c;

}

8、多边形扫面线填充

①原理：

扫描线多边形区域填充算法是按扫描线顺序，计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的像素。区间的端点可以通过计算扫描线与多边形边界线的交点获得。对于一条扫描线，多边形的填充过程可以分为4个步骤。

（1）求交：计算扫描线与多边形各边的交点。

（2）排序：把所有交点按x值递增顺序排序。

（3）配对：第一个与第二个，第三个与第四个等，每对交点代表扫描线与多边形的一个相交区间。

（4）填色：把相交区间内的像素置成多边形颜色，把相交区间外的像素置成背景色。

②代码实现：

void CMyView::OnScanfill() //扫描线算法进行多边形区域填充，如图6所示

{

CDC\* pDC=GetDC();

CPen newpen(PS\_SOLID,1,RGB(0,255,0));

CPen \*old=pDC->SelectObject(&newpen);

int j,k,s=0;

int p[5]; //每根扫描线交点

int pmin,pmax;

for(int i=0;i<6;i++)//建立边表

{

edge[i].dx=(float)(spt[i+1].x-spt[i].

x)/(spt[i+1].y-spt[i].y);

if(spt[i].y<=spt[i+1].y){

edge[i].num=i;

edge[i].ymin=spt[i].y;

edge[i].ymax=spt[i+1].y;

edge[i].xmin=(float)spt[i].x;

edge[i].xmax=(float)spt[i+1].x;

pmax=spt[i+1].y;

pmin=spt[i].y;

}

else{

edge[i].num=i;

edge[i].ymin=spt[i+1].y;

edge[i].ymax=spt[i].y;

edge[i].xmax=(float)spt[i].x;

edge[i].xmin=(float)spt[i+1].x;

pmax=spt[i].y;

pmin=spt[i+1].y;

}

}

for(int r=1;r<6;r++) //排序edge(yUpper，xIntersect)

{

for(int q=0;q<6-r;q++)

{

if(edge[q].ymin<edge[q+1].ymin)

{

newedge[0]=edge[q]; edge[q]=edge[q+1];

edge[q+1]=newedge[0];

}

}

}

for(int scan=pmax-1;scan>pmin+1;scan--)

{

int b=0;

k=s;

for(j=k;j<6;j++)

{

if((scan>edge[j].ymin)&&(scan<=edge[j].ymax))//判断与线段相交

{

if(scan==edge[j].ymax)

{

if(spt[edge[j].num+1].y<edge[j].ymax)

{

b++;

p[b]=(int)edge[j].xmax;

}

if(spt[edge[j].num-1].y<edge[j].ymax)

{

b++;

p[b]=(int)edge[j].xmax;

}

}

if((scan>edge[j].ymin)&&(scan<edge[j].ymax))

{

b++;

p[b]=(int)(edge[j].xmax+edge[j].dx\*(scan-edge[j].

ymax));

}

}

//pDC->LineTo(spt[edge[0].num].x,spt[edge[0].num].y);

if(scan<=edge[j].ymin)//

s=j;

}

if(b>1)

{

for(int u=1;u<b;u++)

{

pDC->MoveTo(p[u]+3,scan);

u++;

pDC->LineTo(p[u],scan);

}

}

}

pDC->SelectObject(old);

}

9、种子扫面线填充

①原理：

算法的基本过程如下：给定种子点（x，y），首先填充种子点所在扫描线上给定区域的一个区段，然后确定与这一区段相连通的上、下两条扫描线上位于给定区域内的区段，并依次保存下来。反复这个过程，直到填充结束。

区域填充的扫描线算法可由下列3个步骤实现。

（1）初始化：确定种子点元素（x，y）。

（2）判断种子点（x，y）是否满足非边界、非填充色的条件，若满足条件，以y作为当前扫描线沿当前扫描线向左、右两个方向填充，直到边界。

（3）确定新的种子点：检查与当前扫描线y上、下相邻的两条扫描线上的像素。若存在非边界、未填充的像素，则返回步骤（2）进行扫描填充。直至区域所有元素均为填充色，程序结束。

②代码实现：

void CMyView::OnSeedfill() //种子算法进行多边形区域填充，如图7所示

{

CWindowDC dc (this);

int fill=RGB(0,255,0);//填充色

int boundary=RGB(255,0,0);//边界颜色

CPoint pt=s\_point;

int x,y,p0,pmin,pmax;

//求多边形的最大最小值（y值）

for(int m=1;m<7;m++)

{

for(int n=0;n<7-m;n++)

{

if(spt[n].y<spt[n+1].y)

{

p0=spt[n].y;

spt[n]=spt[n+1];

spt[n+1]=p0;

}

}

}

pmax=spt[0].y,pmin=spt[6].y;

//设置种子点坐标

x=s\_point.x;

y=s\_point.y;

//从最大边的下一条边开始，即边界里面第一条边

for(;y<pmax+1;y++)

{

int current=dc.GetPixel(x,y); //获取种子点颜色值

//填充边界内未被填充的点

while((current!=boundary)&&(current!=fill))//判断种子点是否为边界点，是否被填充

{

dc.SetPixel(x,y,fill);//填充

x++; //继续下一个点（右边）

current=dc.GetPixel(x,y); //获取点的颜色值

}

//继续种子点左侧（与上相同）

x=s\_point.x;

x--;

current=dc.GetPixel(x,y);

while((current!=boundary)&&(current!=fill))

{

dc.SetPixel(x,y,fill);

x--;

current=dc.GetPixel(x,y);

}

x=s\_point.x;

}

//设置下一个种子点（当前种子点对应的下一行的点）

//与上面思路相同

x=s\_point.x;

y=s\_point.y-1;

for(;y>pmin-2;y--)

{

int current=dc.GetPixel(x,y);

while((current!=boundary)&&(current!=fill))

{

dc.SetPixel(x,y,fill);

x++;

current=dc.GetPixel(x,y);

}

x=s\_point.x;

x--;

current=dc.GetPixel(x,y);

while((current!=boundary)&&(current!=fill))

{

dc.SetPixel(x,y,fill);

x--;

current=dc.GetPixel(x,y);

}

x=s\_point.x;

}

}

10、绘制多面体模块

①原理：通过调用OpenGL中的绘图函数，实现绘制图形。

②代码实现：

switch(iMode)

{

case 1:

glutWireTetrahedron();

break; //绘制线框正四面体

case 2:

glutSolidTetrahedron();

break; //绘制实体正四面体

case 3:

glutWireOctahedron();

break; //绘制线框正八面体

case 4:

glutSolidOctahedron();

break; //绘制实体正八面体

case 5:

glutWireSphere(1.0f,15,15);

break; //绘制线框球

case 6:

glutSolidSphere(1.0f,15,15);

break; //绘制实体球

case 7:

glutWireTeapot(1.0f);

break; //绘制线框茶壶

case 8:

glutSolidTeapot(1.0f);

break; //绘制实体茶壶

case 9:

gluSphere(obj,1.0f,15,15);

break; //绘制二次曲面（球）

case 10:

gluCylinder(obj,1.0f,0.0f,1.0f,15,15);

break; //绘制二次曲面（圆锥）

case 11:

gluPartialDisk(obj,0.3f,0.8f,15,15,30.0f,260.0f);

break; //绘制二次曲面（圆环）

default:

break;

}

11、光滑明暗处理模块

①原理：使用openGl绘制图形。

②代码实现：

void CCMyDrawView::Onsimplemoxing()

{

auxInitDisplayMode (AUX\_SINGLE | AUX\_RGBA);

auxInitPosition (0, 0, 500, 500);//初始化窗口位置

auxInitWindow ("光滑明暗处理");//设置标题

myinit();//初始化

auxReshapeFunc (myReshape);//窗口改变函数

auxMainLoop(display);//定义绘制循环函数

}

void myinit (void)

{

glShadeModel (GL\_SMOOTH);//设置默认着色模式

}

void object(void)

{

glBegin (GL\_POLYGON)

//设置顶点坐标

　　//设置当前颜色

glColor3f (1.0, 0.0, 0.0);

glVertex2f (4.0, 4.0);

glColor3f(1.0,1.0,1.0);

glVertex2f (12.0, 4.0);

glColor3f(0.0,0.0,1.0);

glVertex2f (12.0, 12.0);

glColor3f(0.0,1.0,0.0);

glVertex2f (4.0, 12.0);

glEnd ();

}

void CALLBACK display(void)

{

glClear (GL\_COLOR\_BUFFER\_BIT);//清楚缓冲区

object ();

glFlush ();//刷新缓冲

}

void CALLBACK myReshape(GLsizei w, GLsizei h)

{

glViewport(0, 0, w, h);//创建窗口

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();//重置当前指定的矩阵为单位矩阵

if (w <= h) //定义一个二维图像投影矩阵

gluOrtho2D (0.0, 16.0, 0.0, 16.0 \* (GLfloat) h/(GLfloat) w);

else

gluOrtho2D (0.0, 16.0 \* (GLfloat) w/(GLfloat) h, 0.0, 16.0);

glMatrixMode(GL\_MODELVIEW);//指定当前矩阵

}

12、简单的光照模型

①原理：使用openGl绘制图形。

②代码实现：

void CCMyDrawView::Onputongguangzhao()

{

auxInitDisplayMode (AUX\_SINGLE | AUX\_RGBA); //窗口显示单缓存和RGB(彩色)模式

auxInitPosition (0, 0, 500, 500);//定义窗口的初始位置

auxInitWindow ("简单的光照模型");//窗口标题

myinit();//初始化

auxReshapeFunc (myReshape);//窗口变化函数

auxMainLoop(display);//定义绘制循环函数

}

void myinit(void)//初始化

{

GLfloat light\_position[] = { 1.0, 1.0, 1.0, 0.0 };//定义第0个光源

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);//指定第0个光源的位置

glEnable(GL\_LIGHTING);//启用灯源

glEnable(GL\_LIGHT0);//启用0号灯

glDepthFunc(GL\_LESS);//指定深度缓冲比较值,如果输入的深度值小于参考值，则通过

glEnable(GL\_DEPTH\_TEST);//启用深度测试。

}

void CALLBACK display(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);//清除缓冲

auxSolidSphere(1.0);//绘制球

glFlush();//刷新

}

void CALLBACK myReshape(GLsizei w, GLsizei h)

{

glViewport(0, 0, w, h);//创建窗口

glMatrixMode(GL\_PROJECTION);//指定当前矩阵

glLoadIdentity();//重置当前指定的矩阵为单位矩阵

if (w <= h)//当前的可视空间设置为正投影空间

glOrtho (-1.5, 1.5, -1.5\*(GLfloat)h/(GLfloat)w,

1.5\*(GLfloat)h/(GLfloat)w, -10.0, 10.0);

else

glOrtho (-1.5\*(GLfloat)w/(GLfloat)h,

1.5\*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);

glMatrixMode(GL\_MODELVIEW);//指定当前矩阵

glLoadIdentity();/重置当前指定的矩阵为单位矩阵

}

13、真实感图形模块

①原理：使用openGl绘制图形。

②代码实现：

void CCMyDrawView::Onzhenshigantuxing()

{

auxInitDisplayMode (AUX\_SINGLE | AUX\_RGBA | AUX\_DEPTH16);

auxInitPosition (0, 0, 500, 500);//初始化窗口位置

auxInitWindow ("材质处理例子");//设置标题

myinit();//初始化

auxReshapeFunc (myReshape);//窗口改变函数

auxMainLoop(display);//定义绘制循环函数

}

void myinit(void)

{

GLfloat mat\_ambient[]= { 0.8, 0.8, 0.8, 1.0 };

GLfloat mat\_diffuse[]= { 0.8, 0.0, 0.8, 1.0 }; /\* 紫色 \*/

GLfloat mat\_specular[] = { 1.0, 0.0, 1.0, 1.0 };

GLfloat mat\_shininess[] = { 50.0 };

GLfloat light\_diffuse[]= { 0.0, 0.0, 1.0, 1.0}; /\* 蓝色 \*/

GLfloat light\_position[] = { 1.0, 1.0, 1.0, 0.0 };

//指定用于光照计算的当前材质属性

glMaterialfv(GL\_FRONT, GL\_AMBIENT, mat\_ambient);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse);

glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular);

glMaterialfv(GL\_FRONT, GL\_SHININESS, mat\_shininess);

/指定光源的位置

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse);

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);

glEnable(GL\_LIGHTING);//启用灯源

glEnable(GL\_LIGHT0);//启用0号灯

glDepthFunc(GL\_LESS);//指定深度缓冲比较值,如果输入的深度值小于参考值，则通过

glEnable(GL\_DEPTH\_TEST);//启用深度测试。

}

void CALLBACK display(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);//清楚缓冲

auxSolidSphere(1.0);//绘制球

glFlush();//刷新

}

void CALLBACK myReshape(GLsizei w, GLsizei h)

{

glViewport(0, 0, w, h);//创建窗口

glMatrixMode(GL\_PROJECTION);//指定当前矩阵

glLoadIdentity();//重置当前指定的矩阵为单位矩阵

if (w <= h)//定义一个二维图像投影矩阵

glOrtho (-1.5, 1.5, -1.5\*(GLfloat)h/(GLfloat)w,

1.5\*(GLfloat)h/(GLfloat)w, -10.0, 10.0);

else

glOrtho (-1.5\*(GLfloat)w/(GLfloat)h,

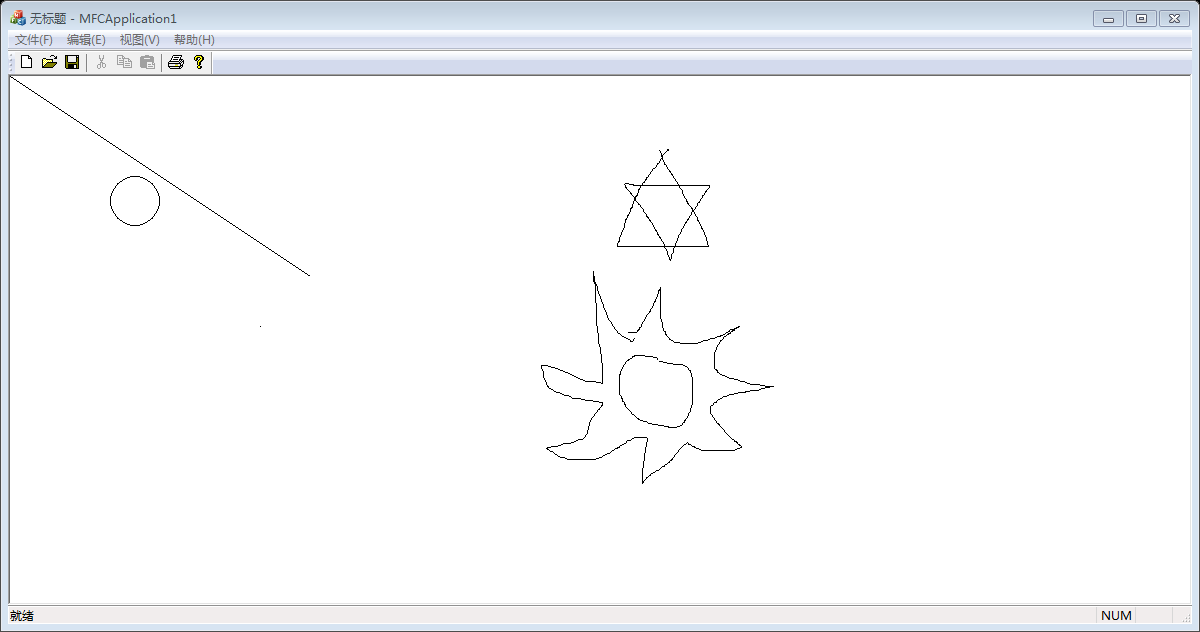
1.5\*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);

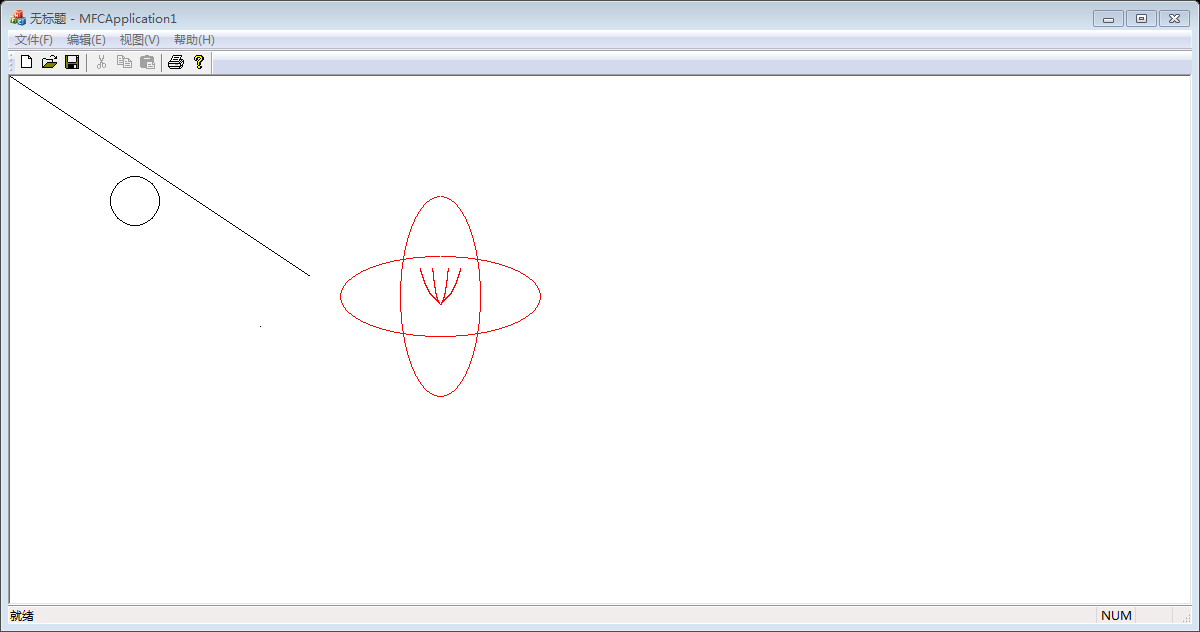
glMatrixMode(GL\_MODELVIEW);//指定当前矩阵

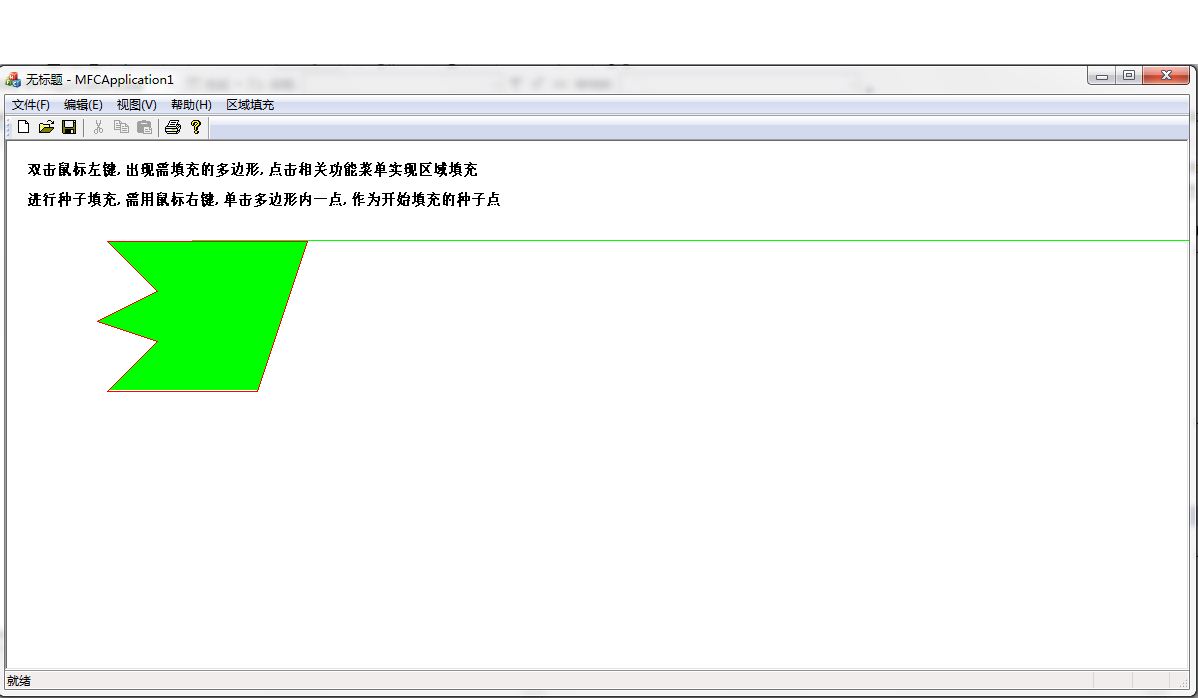
glLoadIdentity();//重置当前指定的矩阵为单位矩阵

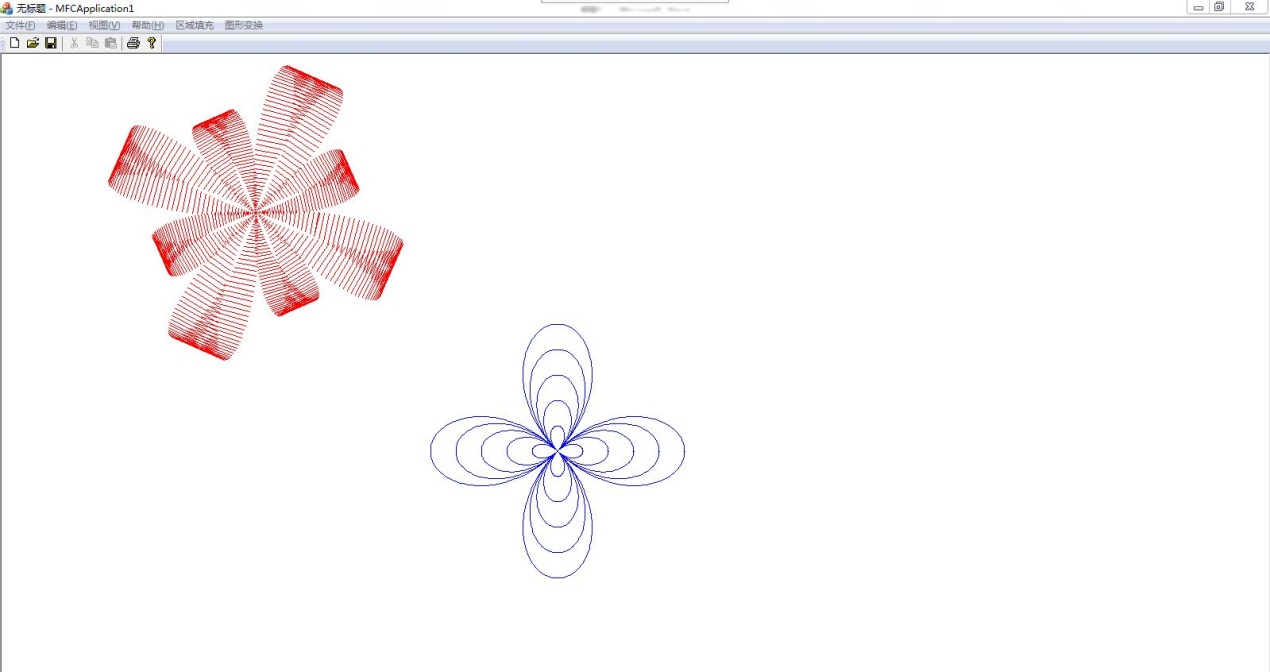
}

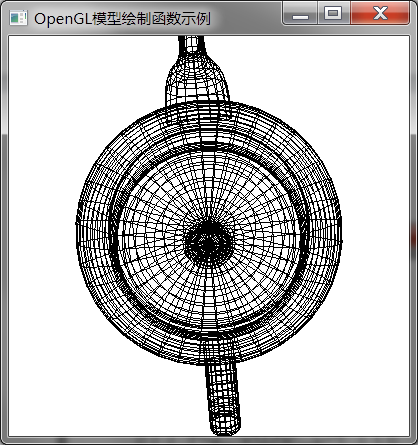
四、 系统测试：











五、总结 ：

了解了图形学领域的流行算法。学习使用了OPENGL库中提供的多种绘制函数。

也能够理解在平时显示器显示效果异常时，可能出现的原因。也学习到了在处理复杂问题时，简化问题的方法和提升算法效率的方法，例如编码区块等。也熟练了VisualStudioIDE的很多功能。